String Matching

The string matching problem is as follows:

Given a pattern p, and a string of text t, determine if p appears as a substring of t or not. Also, an auxiliary problem is to identify where each occurence of p is located in t.

The first way most people would think of solving this problem is by using brute force. The idea is as follows:

Try to match the pattern with the text at EACH possible location. When attempting a single match, compare letters from the beginning, stopping when you hit a discrepancy.

Using this basic idea, we can come up with some code to implement this brute force algorithm:

```
public static boolean match(String text, String pattern) {

   for (int i=0; i<= text.length()-pattern.length(); i++) {

      int j=0;
      while (j < pattern.length() &&
            pattern.charAt(j) == text.charAt(i+j))
         j++;
      if (j == pattern.length())
         return true;
   }
   return false;
}
```

Boyer-Moore Algorithm

This algorithm does a couple things differently than the brute force algorithm:

1) Instead of checking for a string match from the beginning of the string, check for the match from the end. This allows jumps of more than one space when hitting a discrepancy.

2) Instead of always moving ahead one space after you have eliminated a string choice, jump several spaces if you can.

Let's go through an example:

```
      0          1          2          3
Text: abbac cabca ccbba abacb babbb bbaab bbabb baaaa
Pattern: bbaab
```

When we first line the pattern up with the beginning of the text, when we look at the last characters, b and c do not match. In this case, we can shift the string over until we get a character in the pattern to match the c. Since there is NO c in the pattern, we can simply shift over the pattern to compare with the substring from array index 5 to 9 of the Text.

Since a doesn't match b, we will slide the pattern over until we find an a to match the a at array index 9. This time, that means only sliding over the pattern by one spot. This time we are comparing the b with the c in array index 10 of the Text. So, once again we get to slide over 5 spots.

Now, when we compare, we see that the a from the text does NOT match the b from the pattern, so we move over one. Finally at this point we get our first match.

Boyer-Moore Details

The key to the implementation is knowing exactly how many spaces to "slide over" whenever you two characters do not match. You can actually conduct a small amount of precomputation to save time with this determination. In particular, you must calculate the index of the last occurence of all characters in the pattern to be matched. For example, with the pattern bbaab, the last occurence of a is index 3 and the last occurence of b is index 4. In the worst case you shift over one spot. But, in other cases, you shift such that the current character (where there was a discrepancy) gets matched with the last occurence of that character in the pattern. Here is an example of the latter case:

Text: c o m p u t e r s c i e n c e t w o
Pattern: t r a i l s
t r a i l s

First we match the 's' in trails to the 't' in computer. Since there is no match, we will "slide" trails over until the 't' in trails matches the t in computer and then start our comparison with 's' in trails and 'i' in science.

Now, let's look at a case where you only get to slide one place:

Text: b i g t e s t i s t o m o r r o w
Pattern: t e m p e s t
t e m p e s t

Here, we will have the t's match, followed by the s's and e's. The discrepancy is when we have the t in "bigtest" differ from the p in "tempest." Since the discrepancy occurs at the t, we are supposed to shift the pattern to the last t in the pattern. Notice that this shift is actually backwards!!! (Since the last t in the pattern is in the last spot of the pattern.)

One might say that we really know enough to shift until the first 't' in "tempest" lines up with the t in "bigtest", but then we would have to preprocess more imformation. This cost in preprocessing is simply not worth it.

The algorithm is included in the text, so I won't reproduce it here, except to mention what each variable in the text's algorithm keeps track of and comments on certain steps of the algorithm:

i: current index to text
j: current index to pattern

if j is 0, that means that each character has successfully been compared from the back.

On a mismatch, if j < 1 + last(T[i]), then we end up moving the pattern over one spot. Otherwise, we may jump.

Knuth-Morris-Pratt Algorithm

The crux to the efficiency of this algorithm is its failure function. It allows us not to waste prior comparisons done when we happened to line up the pattern in a particular location and had some but not all the letters match.

This failure function f(j) computes the longest prefix of the pattern P that is a *suffix* of P[1..j], where P[1..j] is the substring of P from array index 1 through j inclusive.

For example, for the pattern "abacab" we have the following failure function:

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

f(2) is 1, because the prefix "a" is a suffix of "aba"
f(4) is 1, because the prefix "a" is a suffix of "abaca"
f(5) is 2, because the prefix "ab" is a suffix of "abacab"

The idea is to test for a match going forward (like the brute force algorithm, but unlike the Boyer-Moore algorithm), but whenever a failure is found, we NEVER move the index to the text backwards, we can either leave it in the same place or move it forwards. Instead, we successively move the pattern forward to test. Furthermore, if the failure function is positive, then we do NOT need to check the beginning letters of the pattern because *we know they match already!*

Here is a trace of the algorithm:

Use pattern "babbac", then we have the following failure function:

j	0	1	2	3	4	5
P[j]	b	a	b	b	a	c
f(j)	0	0	1	1	2	0

Text: abaca abacc ababb ac

a	b	a	b	a	a	b	a	c	c	a	b	a	b	b	a	c
b	a	b	b	a	c											
	b	a	b	b	a	c										
			b	a	b	b	a	c								
					b	a	b	b	a	c						
						b	a	b	b	a	c					
								b	a	b	b	a	c			
									b	a	b	b	a	c		
										b	a	b	b	a	c	
											b	a	b	b	a	c

Here is a look at the pseudocode for the algorithm:

```
P is pattern of length m, T is text of length n
i = 0, j = 0
while (i < n) {
        if (P[j] == T[i])
                if (j == m-1)
                        return i-m+1, for beginning index of match
                i = i + 1
                j = j + 1
        else if (j > 0)
                j = failure_function(j-1)
        else
                i = i + 1
}
return no substring
```

Notice that i is never decremented. There are only situations where it is not incremented. (This is when a discrepancy is found after some matching characters.) j is incremented when we are checking successive letters for a match, but is set to the failure function when we have hit a discrepancy after some matching letters. In a few cases, this allows for us skipping some comparisons, if a prefix of the pattern is a suffix of the substring tried.

In any event, we never run more than two comparisons on the same letter in the text. Thus, this portion of the algorithm runs in O(n) time. Computing the failure function takes O(m) time, thus the total running time is O(n+m). Of course, one could argue that m < n is always true, otherwise, the pattern can not be found. Thus, O(n+m) simplifies to O(n) in that case.